BOOK 9: ASYNC JAVASCRIPT
Chapter 1: Understanding Async

SYNCHRONOUS CODE

Normal code runs line by line:

```
console.log("First")
console.log("Second")
console.log("Third")
```

Output:
First
Second
Third

Each line waits for the previous line to finish.

THE PROBLEM WITH SYNC

Some operations take time:

- Fetching data from a server
- Reading a large file
- Waiting for user input
- Timers and delays

If JavaScript waited for these, the page would freeze.

Imagine clicking a button and nothing works while data loads.

Unacceptable.

ASYNCHRONOUS CODE

Async code does not block. It starts an operation and continues.

```
console.log("Start")

setTimeout(function() {
 console.log("Timer finished")
}, 2000)

console.log("End")
```

Output:
Start
End
Timer finished

The timer starts, but JavaScript continues to "End" immediately.

After 2 seconds, "Timer finished" appears.

HOW ASYNC WORKS

JavaScript has an event loop. It processes code in order but handles async operations differently.

1. Run synchronous code first
2. Put async callbacks in a queue
3. When sync code finishes, process the queue

Think of it like a restaurant:
- You order food (start async operation)
- You continue your conversation (run other code)
- When food is ready, waiter brings it (callback runs)

SETTIMEOUT

Run code after a delay:

```
setTimeout(function() {
 console.log("This runs after 1 second")
}, 1000)
```

First argument: the function to run
Second argument: delay in milliseconds

1000 milliseconds = 1 second

```
setTimeout(function() {
 console.log("2 seconds passed")
}, 2000)

setTimeout(function() {
 console.log("Half a second passed")
}, 500)
```

Multiple timers run independently.

SETINTERVAL

Run code repeatedly:

```
setInterval(function() {
 console.log("This runs every second")
}, 1000)
```

This never stops on its own. You must clear it.

```
let count = 0
```

```
let intervalId = setInterval(function() {
 count++
 console.log("Count:", count)

 if (count >= 5) {
 clearInterval(intervalId)
 console.log("Stopped")
 }
}, 1000)
```

CLEARING TIMERS

clearTimeout cancels a timeout:

```
let timeoutId = setTimeout(function() {
 console.log("This will not run")
}, 5000)

clearTimeout(timeoutId)
```

clearInterval cancels an interval:

```
let intervalId = setInterval(function() {
 console.log("Tick")
}, 1000)

setTimeout(function() {
 clearInterval(intervalId)
 console.log("Interval stopped")
}, 5000)
```

ZERO DELAY

Even with 0 delay, setTimeout is async:

```
console.log("First")

setTimeout(function() {
 console.log("Timeout")
}, 0)

console.log("Second")
```

Output:
First
Second
Timeout

The timeout callback goes to the queue and waits for sync code to finish.

NESTED TIMEOUTS

Create a sequence of delays:

```
setTimeout(function() {
 console.log("First")

 setTimeout(function() {
 console.log("Second")

 setTimeout(function() {
 console.log("Third")
 }, 1000)
 }, 1000)
}, 1000)
```

Each message appears 1 second apart.

This works but gets messy. We will learn better ways.

PRACTICAL EXAMPLES

Countdown timer:

```
function countdown(seconds) {
 let remaining = seconds

 let intervalId = setInterval(function() {
 console.log(remaining)
 remaining--

 if (remaining < 0) {
 clearInterval(intervalId)
 console.log("Done!")
 }
 }, 1000)
}

countdown(5)
```

Delayed message:

```
function showMessageLater(message, delay) {
 setTimeout(function() {
 alert(message)
 }, delay)
}

showMessageLater("Hello!", 3000)
```

Auto-save feature:

```
let saveTimeout = null

function scheduleAutoSave() {
 if (saveTimeout) {
 clearTimeout(saveTimeout)
```

```
 }

 saveTimeout = setTimeout(function() {
 console.log("Auto-saving...")
 saveData()
 }, 5000)
}

document.addEventListener("input", scheduleAutoSave)
```

Polling for updates:

```
function pollForUpdates() {
 setInterval(function() {
 console.log("Checking for updates...")
 checkForUpdates()
 }, 30000)
}
```

THE CALLBACK PATTERN

A callback is a function passed to another function:

```
function doSomethingAsync(callback) {
 setTimeout(function() {
 let result = "Operation complete"
 callback(result)
 }, 1000)
}

doSomethingAsync(function(result) {
 console.log(result)
})
```

The callback runs when the async operation finishes.

This is the foundation of async JavaScript.

PRACTICE

Use setTimeout to delay code execution.

Create a countdown using setInterval.

Clear a timer before it completes.

Understand the order of sync vs async code.

Write a function that takes a callback.

WHAT YOU LEARNED

You learned synchronous code blocks.
You learned asynchronous code does not block.
You learned setTimeout runs code after a delay.
You learned setInterval runs code repeatedly.
You learned to clear timers with clearTimeout and clearInterval.
You learned callbacks run when async operations finish.

The next chapter teaches you about callback patterns.
BOOK 9: ASYNC JAVASCRIPT
Chapter 2: Callbacks

WHAT IS A CALLBACK

A callback is a function passed to another function:

```
function greet(name, callback) {
 let message = "Hello, " + name
 callback(message)
}

greet("Alice", function(message) {
 console.log(message)
})
```

Output: Hello, Alice

The callback is called after some work is done.

CALLBACKS FOR ASYNC OPERATIONS

Callbacks handle results of async operations:

```
function fetchUser(userId, callback) {
 setTimeout(function() {
 let user = { id: userId, name: "Alice" }
 callback(user)
 }, 1000)
}

fetchUser(1, function(user) {
 console.log("User loaded:", user.name)
})
```

The callback runs when the data is ready.

ERROR-FIRST CALLBACKS

Node.js convention: first argument is error, second is result:

```
function readFile(filename, callback) {
 setTimeout(function() {
 if (filename === "") {
 callback(new Error("Filename required"), null)
 } else {
 callback(null, "File contents here")
 }
 }, 1000)
}

readFile("document.txt", function(error, contents) {
 if (error) {
 console.log("Error:", error.message)
 return
 }
 console.log("Contents:", contents)
})
```

Always check for error first.

CALLBACK HELL

Nested callbacks become hard to read:

```
getUser(userId, function(user) {
 getOrders(user.id, function(orders) {
 getOrderDetails(orders[0].id, function(details) {
 getShippingInfo(details.shippingId, function(shipping) {
 console.log("Shipping:", shipping)
 })
 })
 })
})
```

This pyramid shape is called "callback hell" or "pyramid of doom".

Each operation depends on the previous result.

PROBLEMS WITH CALLBACK HELL

Hard to read and understand.
Hard to handle errors.
Hard to maintain and modify.
Hard to add new operations in the middle.

Error handling makes it worse:

```
getUser(userId, function(error, user) {
 if (error) {
 handleError(error)
 return
 }
 getOrders(user.id, function(error, orders) {
 if (error) {
 handleError(error)
 return
 }
 getOrderDetails(orders[0].id, function(error, details) {
 if (error) {
 handleError(error)
 return
 }
 console.log("Details:", details)
 })
 })
})
```

FLATTENING CALLBACKS

Break callbacks into named functions:

```
function handleUser(error, user) {
 if (error) {
 handleError(error)
 return
 }
 getOrders(user.id, handleOrders)
}

function handleOrders(error, orders) {
 if (error) {
 handleError(error)
 return
 }
 getOrderDetails(orders[0].id, handleDetails)
}

function handleDetails(error, details) {
 if (error) {
 handleError(error)
 return
 }
 console.log("Details:", details)
}

getUser(userId, handleUser)
```

Flatter, but still awkward. Variables are harder to share.

PRACTICAL CALLBACK EXAMPLES

Load data step by step:

```
function loadUserProfile(userId, callback) {
 setTimeout(function() {
 let user = { id: userId, name: "Alice" }
 callback(null, user)
 }, 500)
```

```
}

function loadUserPosts(userId, callback) {
 setTimeout(function() {
 let posts = [
 { id: 1, title: "First Post" },
 { id: 2, title: "Second Post" }
 ]
 callback(null, posts)
 }, 500)
}

function loadPostComments(postId, callback) {
 setTimeout(function() {
 let comments = [
 { id: 1, text: "Great post!" },
 { id: 2, text: "Thanks for sharing" }
 ]
 callback(null, comments)
 }, 500)
}

loadUserProfile(1, function(error, user) {
 if (error) return console.log(error)

 console.log("User:", user.name)

 loadUserPosts(user.id, function(error, posts) {
 if (error) return console.log(error)

 console.log("Posts:", posts.length)

 loadPostComments(posts[0].id, function(error, comments) {
 if (error) return console.log(error)

 console.log("Comments:", comments.length)
 })
 })
})
```

ARRAY OF CALLBACKS

Process items one by one:

```
function processItems(items, processItem, callback) {
 let results = []
 let index = 0

 function next() {
 if (index >= items.length) {
 callback(null, results)
 return
 }

 let item = items[index]
 processItem(item, function(error, result) {
 if (error) {
 callback(error, null)
 return
 }

 results.push(result)
 index++
 next()
 })
 }

 next()
}

let numbers = [1, 2, 3, 4, 5]

function doubleAsync(n, callback) {
 setTimeout(function() {
 callback(null, n * 2)
 }, 100)
}
```

```
processItems(numbers, doubleAsync, function(error, results) {
 console.log(results)
})
```

Output: [2, 4, 6, 8, 10]

PARALLEL CALLBACKS

Run multiple operations at once:

```
function fetchAll(urls, callback) {
 let results = []
 let completed = 0

 urls.forEach(function(url, index) {
 fetchData(url, function(error, data) {
 if (error) {
 callback(error, null)
 return
 }

 results[index] = data
 completed++

 if (completed === urls.length) {
 callback(null, results)
 }
 })
 })
}

let urls = ["/api/users", "/api/posts", "/api/comments"]

fetchAll(urls, function(error, results) {
 if (error) {
 console.log("Error:", error)
```

```
 return
 }
 console.log("All data loaded:", results)
})
```

THE LIMITATIONS

Callbacks work but have issues:

- Inversion of control (you pass control to another function)
- Hard to return values
- Hard to coordinate multiple async operations
- Error handling is manual and repetitive
- Code structure becomes complex

Promises solve these problems.

PRACTICE

Write a function that takes a callback.

Handle errors with error-first callbacks.

Create a sequence of async operations.

Flatten callback hell with named functions.

Understand why callbacks are limiting.

WHAT YOU LEARNED

You learned callbacks are functions passed to other functions.
You learned error-first callback convention.
You learned callback hell is nested callbacks.
You learned to flatten callbacks with named functions.
You learned callbacks have limitations.

The next chapter teaches you about Promises.

BOOK 9: ASYNC JAVASCRIPT

Chapter 3: Promises

WHAT IS A PROMISE

A Promise represents a value that will be available later.

```
let promise = new Promise(function(resolve, reject) {
 setTimeout(function() {
 resolve("Success!")
 }, 1000)
})

promise.then(function(result) {
 console.log(result)
})
```

Output (after 1 second): Success!

PROMISE STATES

A Promise has three states:

pending - initial state, waiting
fulfilled - operation completed successfully
rejected - operation failed

A promise starts pending and becomes either fulfilled or rejected.

Once settled (fulfilled or rejected), it never changes again.

CREATING A PROMISE

The Promise constructor takes an executor function:

```
let promise = new Promise(function(resolve, reject) {
 let success = true

 if (success) {
 resolve("It worked!")
 } else {
 reject("It failed!")
 }
})
```

resolve(value) - fulfills the promise with value
reject(reason) - rejects the promise with reason

CONSUMING A PROMISE

Use then to handle fulfillment:

```
promise.then(function(result) {
 console.log("Success:", result)
})
```

Use catch to handle rejection:

```
promise.catch(function(error) {
 console.log("Error:", error)
```

```
})
```

Chain them together:

```
promise
 .then(function(result) {
 console.log("Success:", result)
 })
 .catch(function(error) {
 console.log("Error:", error)
 })
```

A REAL EXAMPLE

Simulated data fetch:

```
function fetchUser(userId) {
 return new Promise(function(resolve, reject) {
 setTimeout(function() {
 if (userId > 0) {
 resolve({ id: userId, name: "Alice" })
 } else {
 reject(new Error("Invalid user ID"))
 }
 }, 1000)
 })
}

fetchUser(1)
 .then(function(user) {
 console.log("User:", user.name)
 })
 .catch(function(error) {
 console.log("Error:", error.message)
 })
```

CHAINING PROMISES

then returns a new promise:

```
fetchUser(1)
 .then(function(user) {
 console.log("Got user:", user.name)
 return user.id
 })
 .then(function(userId) {
 console.log("User ID is:", userId)
 return userId * 2
 })
 .then(function(doubled) {
 console.log("Doubled:", doubled)
 })
```

Each then passes its return value to the next then.

RETURNING PROMISES IN CHAINS

```
function fetchUser(userId) {
 return new Promise(function(resolve) {
 setTimeout(function() {
 resolve({ id: userId, name: "Alice" })
 }, 500)
 })
}

function fetchOrders(userId) {
 return new Promise(function(resolve) {
 setTimeout(function() {
 resolve([{ id: 1, item: "Laptop" }])
 }, 500)
 })
```

```
}

function fetchOrderDetails(orderId) {
 return new Promise(function(resolve) {
 setTimeout(function() {
 resolve({ orderId: orderId, total: 999 })
 }, 500)
 })
}

fetchUser(1)
 .then(function(user) {
 console.log("User:", user.name)
 return fetchOrders(user.id)
 })
 .then(function(orders) {
 console.log("Orders:", orders.length)
 return fetchOrderDetails(orders[0].id)
 })
 .then(function(details) {
 console.log("Total:", details.total)
 })
 .catch(function(error) {
 console.log("Error:", error.message)
 })
```

No more callback hell. The chain is flat.

ERROR HANDLING

Errors propagate through the chain:

```
fetchUser(-1)
 .then(function(user) {
 console.log("This does not run")
 return fetchOrders(user.id)
 })
```

```
 .then(function(orders) {
 console.log("This does not run either")
 })
 .catch(function(error) {
 console.log("Caught error:", error.message)
 })
```

If any promise rejects, execution jumps to catch.

FINALLY

Runs regardless of success or failure:

```
fetchUser(1)
 .then(function(user) {
 console.log("User:", user.name)
 })
 .catch(function(error) {
 console.log("Error:", error.message)
 })
 .finally(function() {
 console.log("Done loading")
 })
```

Use finally for cleanup like hiding loading spinners.

PROMISE.ALL

Wait for multiple promises:

```
let promise1 = fetchUser(1)
let promise2 = fetchUser(2)
let promise3 = fetchUser(3)
```

```
Promise.all([promise1, promise2, promise3])
 .then(function(users) {
 console.log("All users:", users)
 })
 .catch(function(error) {
 console.log("One failed:", error.message)
 })
```

Promise.all resolves when all promises resolve.
If any promise rejects, Promise.all rejects immediately.

PROMISE.ALLSETTLED

Wait for all promises to settle (resolve or reject):

```
Promise.allSettled([promise1, promise2, promise3])
 .then(function(results) {
 results.forEach(function(result) {
 if (result.status = = = "fulfilled") {
 console.log("Value:", result.value)
 } else {
 console.log("Error:", result.reason)
 }
 })
 })
```

Never rejects. You get all results.

PROMISE.RACE

Get the first promise to settle:

```
let fast = new Promise(function(resolve) {
 setTimeout(function() {
```

```
 resolve("Fast")
 }, 100)
})

let slow = new Promise(function(resolve) {
 setTimeout(function() {
 resolve("Slow")
 }, 500)
})

Promise.race([fast, slow])
 .then(function(result) {
 console.log("Winner:", result)
 })
```

Output: Winner: Fast

PROMISE.ANY

Get the first promise to fulfill:

```
let fail = new Promise(function(resolve, reject) {
 setTimeout(function() {
 reject("Failed")
 }, 100)
})

let succeed = new Promise(function(resolve) {
 setTimeout(function() {
 resolve("Success")
 }, 200)
})

Promise.any([fail, succeed])
 .then(function(result) {
 console.log("First success:", result)
 })
```

Output: First success: Success

Promise.any ignores rejections unless all reject.

PROMISE.RESOLVE AND PROMISE.REJECT

Create already-settled promises:

```
let resolved = Promise.resolve("Already done")
let rejected = Promise.reject(new Error("Already failed"))

resolved.then(function(value) {
 console.log(value)
})

rejected.catch(function(error) {
 console.log(error.message)
})
```

PRACTICE

Create a promise that resolves after a delay.

Chain multiple then calls.

Handle errors with catch.

Use Promise.all for parallel operations.

Use Promise.race for timeout patterns.

WHAT YOU LEARNED

You learned Promises represent future values.
You learned resolve and reject settle a promise.
You learned then handles success, catch handles errors.
You learned to chain promises for sequences.
You learned Promise.all for parallel operations.
You learned Promise.race for getting the fastest.

The next chapter teaches you async/await.

BOOK 9: ASYNC JAVASCRIPT
Chapter 4: Async/Await

WHAT IS ASYNC/AWAIT

async/await is syntax for working with Promises.

It makes async code look like sync code.

Before (promises):

```
fetchUser(1)
 .then(function(user) {
 return fetchOrders(user.id)
 })
 .then(function(orders) {
 console.log(orders)
 })
```

After (async/await):

```
async function getOrders() {
 let user = await fetchUser(1)
 let orders = await fetchOrders(user.id)
 console.log(orders)
```

```
}

getOrders()
```

Same result. Cleaner code.

ASYNC FUNCTIONS

The async keyword creates an async function:

```
async function getData() {
 return "Hello"
}

getData().then(function(result) {
 console.log(result)
})
```

Output: Hello

An async function always returns a Promise.

If you return a value, it becomes a resolved promise.

THE AWAIT KEYWORD

await pauses until a promise settles:

```
async function example() {
 console.log("Start")

 await new Promise(function(resolve) {
 setTimeout(resolve, 1000)
 })
```

```
 console.log("After 1 second")
}

example()
```

Output:
Start
(1 second pause)
After 1 second

await can only be used inside async functions.

AWAIT WITH PROMISES

Any promise can be awaited:

```
function delay(ms) {
 return new Promise(function(resolve) {
 setTimeout(resolve, ms)
 })
}

async function countdown() {
 console.log(3)
 await delay(1000)
 console.log(2)
 await delay(1000)
 console.log(1)
 await delay(1000)
 console.log("Go!")
}

countdown()
```

ERROR HANDLING

Use try/catch with async/await:

```
async function getUser() {
 try {
 let user = await fetchUser(-1)
 console.log(user)
 } catch (error) {
 console.log("Error:", error.message)
 }
}

getUser()
```

This is cleaner than .catch() chains.

Multiple operations:

```
async function loadData() {
 try {
 let user = await fetchUser(1)
 let orders = await fetchOrders(user.id)
 let details = await fetchOrderDetails(orders[0].id)
 return details
 } catch (error) {
 console.log("Something failed:", error.message)
 return null
 }
}
```

FINALLY WITH TRY/CATCH

```
async function loadWithCleanup() {
 showLoading()
```

```
 try {
 let data = await fetchData()
 displayData(data)
 } catch (error) {
 showError(error)
 } finally {
 hideLoading()
 }
}
```

PARALLEL EXECUTION

Sequential (slow):

```
async function sequential() {
 let user1 = await fetchUser(1)
 let user2 = await fetchUser(2)
 let user3 = await fetchUser(3)

 console.log(user1, user2, user3)
}
```

Each await waits for the previous to finish.

Parallel (fast):

```
async function parallel() {
 let [user1, user2, user3] = await Promise.all([
 fetchUser(1),
 fetchUser(2),
 fetchUser(3)
 ])

 console.log(user1, user2, user3)
}
```

Promise.all starts all fetches at once.

COMBINING AWAIT WITH PROMISE METHODS

```
async function getFirstUser() {
 let user = await Promise.race([
 fetchFromServer1(),
 fetchFromServer2()
 ])
 return user
}

async function getAllData() {
 let results = await Promise.allSettled([
 fetchUsers(),
 fetchPosts(),
 fetchComments()
 ])
 return results
}
```

ASYNC ARROW FUNCTIONS

Arrow functions can be async:

```
let getData = async () => {
 let result = await fetchData()
 return result
}

let process = async (item) => {
 let processed = await processItem(item)
 return processed
}
```

ASYNC METHODS

Class methods can be async:

```
class UserService {
 async getUser(id) {
 let response = await fetch(`/api/users/${id}`)
 let user = await response.json()
 return user
 }

 async getAllUsers() {
 let response = await fetch("/api/users")
 let users = await response.json()
 return users
 }
}

let service = new UserService()
let user = await service.getUser(1)
```

RETURNING VS AWAITING

These are different:

```
async function example1() {
 return fetchData()
}

async function example2() {
 return await fetchData()
}
```

example1 returns a promise immediately.

example2 awaits the result, then returns.

For try/catch, you need await:

```
async function safe() {
 try {
 return await fetchData()
 } catch (error) {
 return null
 }
}
```

LOOPS WITH ASYNC

Sequential processing:

```
async function processAll(items) {
 for (let item of items) {
 await processItem(item)
 console.log("Processed:", item)
 }
}
```

Parallel processing:

```
async function processAllParallel(items) {
 await Promise.all(items.map(async (item) => {
 await processItem(item)
 console.log("Processed:", item)
 }))
}
```

Collecting results:

```
async function fetchAll(ids) {
 let results = []
```

```
 for (let id of ids) {
 let data = await fetchData(id)
 results.push(data)
 }

 return results
}
```

TOP-LEVEL AWAIT

In modules, you can use await at the top level:

```
let config = await loadConfig()
let db = await connectDatabase()

export { config, db }
```

This only works in ES modules.

PRACTICAL EXAMPLES

Retry with async/await:

```
async function fetchWithRetry(url, retries = 3) {
 for (let i = 0; i < retries; i++) {
 try {
 let response = await fetch(url)
 return await response.json()
 } catch (error) {
 console.log(`Attempt ${i + 1} failed`)

 if (i === retries - 1) {
 throw error
 }
```

```
 await delay(1000 * (i + 1))
 }
 }
}
```

Timeout pattern:

```
async function fetchWithTimeout(url, timeout = 5000) {
 let controller = new AbortController()

 let timeoutId = setTimeout(() => controller.abort(), timeout)

 try {
 let response = await fetch(url, { signal: controller.signal })
 return await response.json()
 } finally {
 clearTimeout(timeoutId)
 }
}
```

Sequential API calls:

```
async function getUserProfile(userId) {
 let user = await fetchUser(userId)
 let posts = await fetchPosts(userId)
 let followers = await fetchFollowers(userId)

 return {
 ...user,
 posts,
 followers
 }
}
```

PRACTICE

Create an async function and await a promise.

Use try/catch for error handling.

Execute multiple promises in parallel.

Process an array of items sequentially.

Create a retry function with async/await.

WHAT YOU LEARNED

You learned async functions return promises.
You learned await pauses until a promise settles.
You learned try/catch handles errors.
You learned Promise.all for parallel execution.
You learned async works with arrow functions and methods.
You learned patterns for loops and retries.

The next chapter teaches you about the Fetch API.

BOOK 9: ASYNC JAVASCRIPT

Chapter 5: The Fetch API

WHAT IS FETCH

fetch is the modern way to make HTTP requests:

```
fetch("https://api.example.com/data")
 .then(function(response) {
 return response.json()
 })
 .then(function(data) {
 console.log(data)
```

})

fetch returns a Promise that resolves to a Response object.

FETCH WITH ASYNC/AWAIT

```
async function getData() {
 let response = await fetch("https://api.example.com/data")
 let data = await response.json()
 console.log(data)
}

getData()
```

Much cleaner than XMLHttpRequest.

THE RESPONSE OBJECT

fetch resolves with a Response:

```
async function examineResponse() {
 let response = await fetch("https://api.example.com/data")

 console.log(response.status)
 console.log(response.statusText)
 console.log(response.ok)
 console.log(response.headers)
 console.log(response.url)
}
```

response.status - HTTP status code (200, 404, 500, etc.)
response.statusText - status message ("OK", "Not Found", etc.)
response.ok - true if status is 200-299
response.headers - response headers

response.url - final URL (after redirects)

READING THE BODY

Response body can be read in different formats:

response.json() - parse as JSON
response.text() - read as text
response.blob() - read as binary blob
response.arrayBuffer() - read as ArrayBuffer
response.formData() - read as FormData

These methods return promises:

```
let json = await response.json()

let text = await response.text()

let blob = await response.blob()
```

You can only read the body once. After reading, it is consumed.

CHECKING FOR ERRORS

fetch does not reject on HTTP errors:

```
async function getData() {
 let response = await fetch("https://api.example.com/notfound")

 console.log(response.status)
 console.log(response.ok)
}
```

Output:

404
false

You must check response.ok:

```
async function getData() {
 let response = await fetch("https://api.example.com/data")

 if (!response.ok) {
 throw new Error(`HTTP error: ${response.status}`)
 }

 let data = await response.json()
 return data
}
```

A HELPER FUNCTION

Create a fetch wrapper:

```
async function fetchJSON(url) {
 let response = await fetch(url)

 if (!response.ok) {
 throw new Error(`HTTP error: ${response.status}`)
 }

 return response.json()
}

async function main() {
 try {
 let users = await fetchJSON("https://api.example.com/users")
 console.log(users)
 } catch (error) {
 console.log("Failed:", error.message)
 }
```

```
}
```

POST REQUESTS

Send data with POST:

```
async function createUser(userData) {
 let response = await fetch("https://api.example.com/users", {
 method: "POST",
 headers: {
 "Content-Type": "application/json"
 },
 body: JSON.stringify(userData)
 })

 if (!response.ok) {
 throw new Error(`HTTP error: ${response.status}`)
 }

 return response.json()
}

let newUser = await createUser({
 name: "Alice",
 email: "alice@example.com"
})
```

OTHER HTTP METHODS

PUT request:

```
async function updateUser(id, userData) {
 let response = await fetch(`https://api.example.com/users/${id}`,
{
```

```
 method: "PUT",
 headers: {
 "Content-Type": "application/json"
 },
 body: JSON.stringify(userData)
 })

 return response.json()
}
```

PATCH request:

```
async function patchUser(id, updates) {
 let response = await fetch(`https://api.example.com/users/${id}`,
{
 method: "PATCH",
 headers: {
 "Content-Type": "application/json"
 },
 body: JSON.stringify(updates)
 })

 return response.json()
}
```

DELETE request:

```
async function deleteUser(id) {
 let response = await fetch(`https://api.example.com/users/${id}`,
{
 method: "DELETE"
 })

 return response.ok
}
```

HEADERS

Set request headers:

```
let response = await fetch(url, {
 headers: {
 "Content-Type": "application/json",
 "Authorization": "Bearer " + token,
 "X-Custom-Header": "value"
 }
})
```

Read response headers:

```
let contentType = response.headers.get("Content-Type")
let date = response.headers.get("Date")

response.headers.forEach(function(value, name) {
 console.log(name + ": " + value)
})
```

SENDING FORM DATA

Using FormData:

```
let formData = new FormData()
formData.append("name", "Alice")
formData.append("email", "alice@example.com")

let response = await fetch("https://api.example.com/users", {
 method: "POST",
 body: formData
})
```

From a form element:

```
let form = document.getElementById("myForm")
let formData = new FormData(form)
```

```
let response = await fetch(form.action, {
 method: "POST",
 body: formData
})
```

Do not set Content-Type header with FormData. The browser sets it automatically with the boundary.

UPLOADING FILES

```
let fileInput = document.getElementById("file")
let file = fileInput.files[0]

let formData = new FormData()
formData.append("file", file)

let response = await fetch("https://api.example.com/upload", {
 method: "POST",
 body: formData
})
```

ABORT A REQUEST

Cancel a fetch with AbortController:

```
let controller = new AbortController()

fetch(url, { signal: controller.signal })
 .then(function(response) {
 return response.json()
 })
 .then(function(data) {
 console.log(data)
```

```
 })
 .catch(function(error) {
 if (error.name === "AbortError") {
 console.log("Request was cancelled")
 }
 })

controller.abort()
```

TIMEOUT WITH ABORT

```
async function fetchWithTimeout(url, timeout = 5000) {
 let controller = new AbortController()

 let timeoutId = setTimeout(function() {
 controller.abort()
 }, timeout)

 try {
 let response = await fetch(url, { signal: controller.signal })
 clearTimeout(timeoutId)
 return response
 } catch (error) {
 clearTimeout(timeoutId)
 throw error
 }
}
```

CREDENTIALS

Send cookies with requests:

```
let response = await fetch(url, {
 credentials: "include"
```

})

credentials options:
omit - never send cookies
same-origin - send cookies for same-origin requests (default)
include - always send cookies

PRACTICAL EXAMPLES

Complete API client:

```
class ApiClient {
 constructor(baseUrl) {
 this.baseUrl = baseUrl
 this.token = null
 }

 setToken(token) {
 this.token = token
 }

 async request(endpoint, options = {}) {
 let url = this.baseUrl + endpoint

 let headers = {
 "Content-Type": "application/json",
 ...options.headers
 }

 if (this.token) {
 headers["Authorization"] = "Bearer " + this.token
 }

 let response = await fetch(url, {
 ...options,
 headers
 })
```

```
 if (!response.ok) {
 let error = await response.json()
 throw new Error(error.message || `HTTP ${response.status}`)
 }

 return response.json()
 }

 get(endpoint) {
 return this.request(endpoint)
 }

 post(endpoint, data) {
 return this.request(endpoint, {
 method: "POST",
 body: JSON.stringify(data)
 })
 }

 put(endpoint, data) {
 return this.request(endpoint, {
 method: "PUT",
 body: JSON.stringify(data)
 })
 }

 delete(endpoint) {
 return this.request(endpoint, {
 method: "DELETE"
 })
 }
}

let api = new ApiClient("https://api.example.com")

let users = await api.get("/users")
let newUser = await api.post("/users", { name: "Alice" })
```

PRACTICE

Make a GET request with fetch.

Make a POST request with JSON data.

Handle errors by checking response.ok.

Use AbortController to cancel a request.

Create a reusable API client.

WHAT YOU LEARNED

You learned fetch makes HTTP requests.
You learned to read responses as JSON, text, or blob.
You learned to check response.ok for errors.
You learned POST, PUT, PATCH, DELETE requests.
You learned to send headers and credentials.
You learned AbortController cancels requests.

The next chapter teaches you about handling async errors.
BOOK 9: ASYNC JAVASCRIPT
Chapter 6: Async Error Handling

ERRORS IN PROMISES

Promises can reject:

```
let promise = new Promise(function(resolve, reject) {
 reject(new Error("Something went wrong"))
```

```
})

promise.catch(function(error) {
 console.log("Caught:", error.message)
})
```

Throwing inside a promise executor rejects it:

```
let promise = new Promise(function(resolve, reject) {
 throw new Error("Thrown error")
})

promise.catch(function(error) {
 console.log("Caught:", error.message)
})
```

ERRORS IN THEN

Errors in then handlers are caught by catch:

```
fetchData()
 .then(function(data) {
 throw new Error("Processing failed")
 })
 .catch(function(error) {
 console.log("Caught:", error.message)
 })
```

The error propagates down the chain until caught.

UNHANDLED REJECTIONS

If no catch handles a rejection:

```
let promise = Promise.reject(new Error("Unhandled"))
```

The browser shows an unhandled rejection warning.

Listen for unhandled rejections:

```
window.addEventListener("unhandledrejection", function(event) {
 console.log("Unhandled rejection:", event.reason)
 event.preventDefault()
})
```

Always add catch to your promise chains.

ERRORS WITH ASYNC/AWAIT

Use try/catch:

```
async function getData() {
 try {
 let data = await fetchData()
 return data
 } catch (error) {
 console.log("Error:", error.message)
 return null
 }
}
```

Uncaught errors reject the returned promise:

```
async function getData() {
 let data = await fetchData()
 return data
}

getData().catch(function(error) {
 console.log("Caught outside:", error.message)
})
```

MULTIPLE TRY/CATCH BLOCKS

Handle different operations differently:

```
async function processOrder(orderId) {
 let order

 try {
 order = await fetchOrder(orderId)
 } catch (error) {
 console.log("Failed to fetch order")
 return null
 }

 try {
 await processPayment(order)
 } catch (error) {
 console.log("Payment failed")
 await cancelOrder(orderId)
 return null
 }

 try {
 await sendConfirmation(order)
 } catch (error) {
 console.log("Confirmation failed, but order processed")
 }

 return order
}
```

ERROR TYPES

Create custom error types:

```
class NetworkError extends Error {
 constructor(message) {
 super(message)
 this.name = "NetworkError"
 }
}

class ValidationError extends Error {
 constructor(message, field) {
 super(message)
 this.name = "ValidationError"
 this.field = field
 }
}
```

Handle different error types:

```
async function submitForm(data) {
 try {
 await validateAndSubmit(data)
 } catch (error) {
 if (error instanceof ValidationError) {
 showFieldError(error.field, error.message)
 } else if (error instanceof NetworkError) {
 showNetworkError()
 } else {
 showGenericError()
 }
 }
}
```

RETRYING ON FAILURE

Retry failed operations:

```
async function fetchWithRetry(url, maxRetries = 3) {
 let lastError

 for (let attempt = 1; attempt <= maxRetries; attempt++) {
 try {
 let response = await fetch(url)

 if (!response.ok) {
 throw new Error(`HTTP ${response.status}`)
 }

 return await response.json()
 } catch (error) {
 lastError = error
 console.log(`Attempt ${attempt} failed: ${error.message}`)

 if (attempt < maxRetries) {
 await delay(1000 * attempt)
 }
 }
 }

 throw lastError
}

function delay(ms) {
 return new Promise(function(resolve) {
 setTimeout(resolve, ms)
 })
}
```

EXPONENTIAL BACKOFF

```
async function fetchWithBackoff(url, maxRetries = 5) {
 for (let attempt = 0; attempt < maxRetries; attempt++) {
 try {
 return await fetch(url)
```

```
 } catch (error) {
 if (attempt === maxRetries - 1) {
 throw error
 }

 let waitTime = Math.pow(2, attempt) * 1000
 waitTime += Math.random() * 1000

 console.log(`Waiting ${waitTime}ms before retry`)
 await delay(waitTime)
 }
 }
}
```

Exponential backoff increases wait time with each retry.

FALLBACK VALUES

Return defaults on failure:

```
async function getUser(id) {
 try {
 return await fetchUser(id)
 } catch (error) {
 return {
 id: id,
 name: "Unknown User",
 isDefault: true
 }
 }
}

async function getConfig() {
 try {
 return await fetchConfig()
 } catch (error) {
 console.log("Using default config")
```

```
 return {
 theme: "light",
 language: "en"
 }
 }
}
```

AGGREGATING ERRORS

Collect all errors from parallel operations:

```
async function fetchAllUsers(ids) {
 let results = await Promise.allSettled(
 ids.map(function(id) {
 return fetchUser(id)
 })
 )

 let users = []
 let errors = []

 results.forEach(function(result, index) {
 if (result.status === "fulfilled") {
 users.push(result.value)
 } else {
 errors.push({
 id: ids[index],
 error: result.reason
 })
 }
 })

 if (errors.length > 0) {
 console.log("Some fetches failed:", errors)
 }

 return users
```

```
}
```

CLEANUP WITH FINALLY

Always run cleanup code:

```
async function processWithCleanup() {
 let resource = await acquireResource()

 try {
 await useResource(resource)
 } catch (error) {
 console.log("Processing failed")
 throw error
 } finally {
 await releaseResource(resource)
 console.log("Resource released")
 }
}
```

LOGGING ERRORS

Create an error logging function:

```
async function logError(error, context = {}) {
 let errorInfo = {
 message: error.message,
 stack: error.stack,
 timestamp: new Date().toISOString(),
 ...context
 }

 console.error("Error:", errorInfo)
```

```
 try {
 await fetch("/api/errors", {
 method: "POST",
 headers: { "Content-Type": "application/json" },
 body: JSON.stringify(errorInfo)
 })
 } catch (loggingError) {
 console.error("Failed to log error:", loggingError)
 }
}

async function fetchData() {
 try {
 return await fetch("/api/data")
 } catch (error) {
 await logError(error, { operation: "fetchData" })
 throw error
 }
}
```

PRACTICAL EXAMPLES

Robust API call:

```
async function apiCall(endpoint, options = {}) {
 let {
 retries = 3,
 timeout = 5000,
 fallback = null
 } = options

 let controller = new AbortController()
 let timeoutId = setTimeout(function() {
 controller.abort()
 }, timeout)

 for (let attempt = 1; attempt <= retries; attempt++) {
```

```
 try {
 let response = await fetch(endpoint, {
 signal: controller.signal
 })

 clearTimeout(timeoutId)

 if (!response.ok) {
 throw new Error(`HTTP ${response.status}`)
 }

 return await response.json()
 } catch (error) {
 clearTimeout(timeoutId)

 if (error.name === "AbortError") {
 throw new Error("Request timed out")
 }

 if (attempt === retries) {
 if (fallback !== null) {
 return fallback
 }
 throw error
 }

 await delay(1000 * attempt)
 }
 }
}

let data = await apiCall("/api/users", {
 retries: 3,
 timeout: 5000,
 fallback: []
})
```

PRACTICE

Use try/catch with async/await.

Create custom error types.

Implement retry logic for failed operations.

Handle partial failures with Promise.allSettled.

Add logging to error handlers.

WHAT YOU LEARNED

You learned catch handles promise rejections.
You learned try/catch works with async/await.
You learned to create custom error types.
You learned retry and backoff patterns.
You learned to provide fallback values.
You learned to aggregate errors from parallel operations.

The next chapter teaches you about async patterns.
BOOK 9: ASYNC JAVASCRIPT
Chapter 7: Async Patterns

SEQUENTIAL EXECUTION

Run operations one after another:

```
async function sequential() {
 let result1 = await operation1()
 let result2 = await operation2()
 let result3 = await operation3()
```

```
 return [result1, result2, result3]
}
```

Each operation waits for the previous one to complete.

Use when operations depend on each other.

PARALLEL EXECUTION

Run operations at the same time:

```
async function parallel() {
 let [result1, result2, result3] = await Promise.all([
 operation1(),
 operation2(),
 operation3()
 ])

 return [result1, result2, result3]
}
```

All operations start immediately and run concurrently.

Use when operations are independent.

RACING

Get the first result:

```
async function race() {
 let result = await Promise.race([
 fetchFromServer1(),
 fetchFromServer2()
```

```
 ])

 return result
}
```

Returns as soon as any promise settles.

TIMEOUT PATTERN

Add timeout to any async operation:

```
function withTimeout(promise, ms) {
 let timeout = new Promise(function(resolve, reject) {
 setTimeout(function() {
 reject(new Error("Timeout"))
 }, ms)
 })

 return Promise.race([promise, timeout])
}

async function getData() {
 try {
 let data = await withTimeout(fetchData(), 5000)
 return data
 } catch (error) {
 if (error.message === "Timeout") {
 console.log("Request timed out")
 }
 throw error
 }
}
```

THROTTLING

Limit how often an async operation runs:

```
function throttle(fn, limit) {
 let lastRun = 0

 return async function(...args) {
 let now = Date.now()

 if (now - lastRun < limit) {
 return
 }

 lastRun = now
 return fn.apply(this, args)
 }
}

let throttledFetch = throttle(fetchData, 1000)

throttledFetch()
throttledFetch()
throttledFetch()
```

DEBOUNCING

Wait for inactivity before running:

```
function debounce(fn, delay) {
 let timeoutId = null

 return function(...args) {
 clearTimeout(timeoutId)

 return new Promise(function(resolve) {
 timeoutId = setTimeout(async function() {
 let result = await fn.apply(this, args)
```

```
 resolve(result)
 }, delay)
 })
 }
}

let debouncedSearch = debounce(searchAPI, 300)

async function handleInput(event) {
 let results = await debouncedSearch(event.target.value)
 displayResults(results)
}
```

QUEUE PATTERN

Process items one at a time:

```
class AsyncQueue {
 constructor() {
 this.queue = []
 this.processing = false
 }

 async add(task) {
 return new Promise((resolve, reject) => {
 this.queue.push({ task, resolve, reject })
 this.process()
 })
 }

 async process() {
 if (this.processing) return

 this.processing = true

 while (this.queue.length > 0) {
 let { task, resolve, reject } = this.queue.shift()
```

```
 try {
 let result = await task()
 resolve(result)
 } catch (error) {
 reject(error)
 }
 }

 this.processing = false
 }
}

let queue = new AsyncQueue()

queue.add(async function() {
 return await fetchData(1)
})

queue.add(async function() {
 return await fetchData(2)
})
```

CONCURRENCY LIMIT

Run multiple operations with a limit:

```
async function mapWithLimit(items, limit, fn) {
 let results = []
 let index = 0

 async function worker() {
 while (index < items.length) {
 let currentIndex = index++
 results[currentIndex] = await fn(items[currentIndex])
 }
 }
```

```
 let workers = []
 for (let i = 0; i < Math.min(limit, items.length); i++) {
 workers.push(worker())
 }

 await Promise.all(workers)
 return results
}

let urls = ["/api/1", "/api/2", "/api/3", "/api/4", "/api/5"]

let results = await mapWithLimit(urls, 2, async function(url) {
 return fetch(url).then(r => r.json())
})
```

RETRY WITH BACKOFF

```
async function retryWithBackoff(fn, options = {}) {
 let {
 maxRetries = 3,
 baseDelay = 1000,
 maxDelay = 30000
 } = options

 for (let attempt = 0; attempt < maxRetries; attempt++) {
 try {
 return await fn()
 } catch (error) {
 if (attempt === maxRetries - 1) {
 throw error
 }

 let delay = Math.min(
 baseDelay * Math.pow(2, attempt),
 maxDelay
 )
```

```
 delay += Math.random() * 1000

 await new Promise(r => setTimeout(r, delay))
 }
 }
}

let data = await retryWithBackoff(async function() {
 let response = await fetch("/api/data")

 if (!response.ok) {
 throw new Error("Request failed")
 }

 return response.json()
})
```

CIRCUIT BREAKER

Stop trying after too many failures:

```
class CircuitBreaker {
 constructor(options = {}) {
 this.failureThreshold = options.failureThreshold || 5
 this.resetTimeout = options.resetTimeout || 30000
 this.failures = 0
 this.lastFailure = null
 this.state = "closed"
 }

 async call(fn) {
 if (this.state === "open") {
 if (Date.now() - this.lastFailure > this.resetTimeout) {
 this.state = "half-open"
 } else {
 throw new Error("Circuit is open")
```

```
 }
 }

 try {
 let result = await fn()
 this.onSuccess()
 return result
 } catch (error) {
 this.onFailure()
 throw error
 }
 }

 onSuccess() {
 this.failures = 0
 this.state = "closed"
 }

 onFailure() {
 this.failures++
 this.lastFailure = Date.now()

 if (this.failures >= this.failureThreshold) {
 this.state = "open"
 }
 }
}

let breaker = new CircuitBreaker()

try {
 let data = await breaker.call(async function() {
 return fetchData()
 })
} catch (error) {
 console.log("Failed:", error.message)
}
```

CACHING

Cache async results:

```
function createCache(fn, ttl = 60000) {
 let cache = new Map()

 return async function(key) {
 let cached = cache.get(key)

 if (cached && Date.now() - cached.timestamp < ttl) {
 return cached.value
 }

 let value = await fn(key)

 cache.set(key, {
 value,
 timestamp: Date.now()
 })

 return value
 }
}

let cachedFetch = createCache(async function(url) {
 let response = await fetch(url)
 return response.json()
}, 60000)

let data1 = await cachedFetch("/api/users")
let data2 = await cachedFetch("/api/users")
```

LAZY INITIALIZATION

Initialize only when needed:

```
function lazy(initFn) {
 let instance = null
 let promise = null

 return async function() {
 if (instance) {
 return instance
 }

 if (promise) {
 return promise
 }

 promise = initFn().then(function(result) {
 instance = result
 return result
 })

 return promise
 }
}

let getDatabase = lazy(async function() {
 console.log("Connecting to database...")
 return await connectToDatabase()
})

let db1 = await getDatabase()
let db2 = await getDatabase()
```

PRACTICE

Implement parallel and sequential execution.

Create a timeout wrapper.

Build a debounce function for async operations.

Implement a concurrency limiter.

Create a simple cache for async results.

WHAT YOU LEARNED

You learned sequential vs parallel execution.
You learned timeout and race patterns.
You learned throttling and debouncing.
You learned queue and concurrency limiting.
You learned circuit breaker pattern.
You learned caching async results.

The next chapter teaches you about real-world async applications.
BOOK 9: ASYNC JAVASCRIPT
Chapter 8: Real-World Async

LOADING DATA FOR A PAGE

Load all required data before rendering:

```
async function loadPageData() {
 showLoading()

 try {
 let [user, posts, notifications] = await Promise.all([
 fetchCurrentUser(),
 fetchRecentPosts(),
 fetchNotifications()
 ])
```

```
 renderPage({ user, posts, notifications })
 } catch (error) {
 showError("Failed to load page")
 console.error(error)
 } finally {
 hideLoading()
 }
}

document.addEventListener("DOMContentLoaded", loadPageData)
```

INFINITE SCROLL

Load more data when scrolling:

```
let page = 1
let isLoading = false
let hasMore = true

async function loadMore() {
 if (isLoading || !hasMore) return

 isLoading = true
 showLoadingIndicator()

 try {
 let response = await fetch(`/api/posts?page=${page}`)
 let data = await response.json()

 if (data.posts.length === 0) {
 hasMore = false
 showEndMessage()
 } else {
 appendPosts(data.posts)
 page++
 }
 } catch (error) {
```

```
 showError("Failed to load more posts")
 } finally {
 isLoading = false
 hideLoadingIndicator()
 }
}

window.addEventListener("scroll", function() {
 let scrollBottom = window.scrollY + window.innerHeight
 let documentHeight = document.documentElement.scrollHeight

 if (scrollBottom >= documentHeight - 200) {
 loadMore()
 }
})

loadMore()
```

FORM SUBMISSION

Submit form and handle response:

```
let form = document.getElementById("contact-form")

form.addEventListener("submit", async function(event) {
 event.preventDefault()

 let submitButton = form.querySelector("button[type='submit']")
 submitButton.disabled = true
 submitButton.textContent = "Sending..."

 try {
 let formData = new FormData(form)

 let response = await fetch("/api/contact", {
 method: "POST",
 body: formData
```

```
 })

 if (!response.ok) {
 let error = await response.json()
 throw new Error(error.message)
 }

 showSuccess("Message sent!")
 form.reset()
 } catch (error) {
 showError(error.message)
 } finally {
 submitButton.disabled = false
 submitButton.textContent = "Send"
 }
})
```

SEARCH WITH DEBOUNCE

Search as user types:

```
let searchInput = document.getElementById("search")
let resultsContainer = document.getElementById("results")
let searchTimeout = null

searchInput.addEventListener("input", function() {
 let query = this.value.trim()

 clearTimeout(searchTimeout)

 if (query.length < 2) {
 resultsContainer.innerHTML = ""
 return
 }

 searchTimeout = setTimeout(async function() {
 try {
```

```
 resultsContainer.innerHTML = "<p>Searching...</p>"

 let response = await fetch(`/api/search?q=
${encodeURIComponent(query)}`)
 let results = await response.json()

 displayResults(results)
 } catch (error) {
 resultsContainer.innerHTML = "<p>Search failed</p>"
 }
 }, 300)
})

function displayResults(results) {
 if (results.length === 0) {
 resultsContainer.innerHTML = "<p>No results found</p>"
 return
 }

 resultsContainer.innerHTML = results.map(function(result) {
 return `<div class="result">${result.title}</div>`
 }).join("")
}
```

FILE UPLOAD WITH PROGRESS

Upload file and show progress:

```
let fileInput = document.getElementById("file")
let progressBar = document.getElementById("progress")

fileInput.addEventListener("change", async function() {
 let file = this.files[0]

 if (!file) return

 let formData = new FormData()
```

```
 formData.append("file", file)

 try {
 let response = await uploadWithProgress(
 "/api/upload",
 formData,
 function(percent) {
 progressBar.style.width = percent + "%"
 progressBar.textContent = percent + "%"
 }
 )

 showSuccess("Upload complete!")
 } catch (error) {
 showError("Upload failed")
 }
})

function uploadWithProgress(url, data, onProgress) {
 return new Promise(function(resolve, reject) {
 let xhr = new XMLHttpRequest()

 xhr.upload.addEventListener("progress", function(event) {
 if (event.lengthComputable) {
 let percent = Math.round((event.loaded / event.total) * 100)
 onProgress(percent)
 }
 })

 xhr.addEventListener("load", function() {
 if (xhr.status >= 200 && xhr.status < 300) {
 resolve(JSON.parse(xhr.response))
 } else {
 reject(new Error("Upload failed"))
 }
 })

 xhr.addEventListener("error", function() {
 reject(new Error("Network error"))
 })
```

```
 xhr.open("POST", url)
 xhr.send(data)
 })
}
```

AUTO-SAVE

Save changes automatically:

```
let saveTimeout = null
let lastSavedData = null

function scheduleAutoSave() {
 clearTimeout(saveTimeout)

 saveTimeout = setTimeout(autoSave, 2000)
}

async function autoSave() {
 let currentData = getFormData()

 if (JSON.stringify(currentData) = = =
JSON.stringify(lastSavedData)) {
 return
 }

 showSaving()

 try {
 await fetch("/api/save", {
 method: "POST",
 headers: { "Content-Type": "application/json" },
 body: JSON.stringify(currentData)
 })

 lastSavedData = currentData
```

```
 showSaved()
 } catch (error) {
 showSaveError()
 }
}

document.getElementById("editor").addEventListener("input",
scheduleAutoSave)
```

POLLING FOR UPDATES

Check for new data periodically:

```
let pollInterval = null

function startPolling() {
 if (pollInterval) return

 pollInterval = setInterval(checkForUpdates, 5000)
 checkForUpdates()
}

function stopPolling() {
 clearInterval(pollInterval)
 pollInterval = null
}

async function checkForUpdates() {
 try {
 let response = await fetch("/api/updates?since=" +
lastUpdateTime)
 let data = await response.json()

 if (data.updates.length > 0) {
 processUpdates(data.updates)
 lastUpdateTime = data.timestamp
 }
```

```
 } catch (error) {
 console.log("Failed to check updates:", error)
 }
}

document.addEventListener("visibilitychange", function() {
 if (document.hidden) {
 stopPolling()
 } else {
 startPolling()
 }
})

startPolling()
```

SHOPPING CART

Manage cart with server sync:

```
class Cart {
 constructor() {
 this.items = []
 this.syncing = false
 }

 async load() {
 try {
 let response = await fetch("/api/cart")
 this.items = await response.json()
 this.render()
 } catch (error) {
 console.error("Failed to load cart")
 }
 }

 async addItem(productId, quantity = 1) {
 let tempItem = { productId, quantity, pending: true }
```

```
this.items.push(tempItem)
this.render()

try {
let response = await fetch("/api/cart/items", {
method: "POST",
headers: { "Content-Type": "application/json" },
body: JSON.stringify({ productId, quantity })
})

let item = await response.json()
let index = this.items.indexOf(tempItem)
this.items[index] = item
} catch (error) {
this.items = this.items.filter(i => i !== tempItem)
showError("Failed to add item")
}

this.render()
}

async removeItem(itemId) {
let item = this.items.find(i => i.id === itemId)
item.removing = true
this.render()

try {
await fetch(`/api/cart/items/${itemId}`, {
method: "DELETE"
})

this.items = this.items.filter(i => i.id !== itemId)
} catch (error) {
item.removing = false
showError("Failed to remove item")
}

this.render()
}
```

```
 render() {
 let html = this.items.map(function(item) {
 let classes = []
 if (item.pending) classes.push("pending")
 if (item.removing) classes.push("removing")

 return `<div class="cart-item ${classes.join(' ')}">${item.name}
</div>`
 }).join("")

 document.getElementById("cart").innerHTML = html
 }
}

let cart = new Cart()
cart.load()
```

AUTHENTICATION FLOW

Login and manage auth state:

```
class Auth {
 constructor() {
 this.user = null
 this.token = localStorage.getItem("token")
 }

 async login(email, password) {
 let response = await fetch("/api/login", {
 method: "POST",
 headers: { "Content-Type": "application/json" },
 body: JSON.stringify({ email, password })
 })

 if (!response.ok) {
 let error = await response.json()
 throw new Error(error.message)
```

```
}

let data = await response.json()
this.token = data.token
this.user = data.user

localStorage.setItem("token", this.token)
}

async logout() {
try {
await fetch("/api/logout", {
method: "POST",
headers: { "Authorization": "Bearer " + this.token }
})
} catch (error) {
console.log("Logout request failed")
}

this.token = null
this.user = null
localStorage.removeItem("token")
}

async getUser() {
if (!this.token) return null

if (this.user) return this.user

try {
let response = await fetch("/api/me", {
headers: { "Authorization": "Bearer " + this.token }
})

if (!response.ok) {
this.token = null
localStorage.removeItem("token")
return null
}
```

```
 this.user = await response.json()
 return this.user
 } catch (error) {
 return null
 }
 }

 isLoggedIn() {
 return this.token !== null
 }
}

let auth = new Auth()
```

PRACTICE

Build a page that loads data on startup.

Create an infinite scroll feature.

Implement debounced search.

Build an auto-save feature.

Create an authentication flow.

WHAT YOU LEARNED

You learned to load page data with Promise.all.
You learned infinite scroll patterns.
You learned form submission handling.
You learned debounced search.
You learned auto-save patterns.
You learned polling for updates.

You learned shopping cart with optimistic updates.

The next chapter teaches you about Web Workers.

BOOK 9: ASYNC JAVASCRIPT

Chapter 9: Web Workers

WHAT ARE WEB WORKERS

Web Workers run JavaScript in background threads.

Normal JavaScript runs on the main thread. Heavy computations block the UI.

Workers run separately and do not block.

CREATING A WORKER

Main script (main.js):

```
let worker = new Worker("worker.js")
```

Worker script (worker.js):

```
console.log("Worker started")
```

The worker runs in a separate file.

COMMUNICATING WITH WORKERS

Main script sends messages:

```
let worker = new Worker("worker.js")

worker.postMessage("Hello from main!")

worker.postMessage({ type: "calculate", data: [1, 2, 3] })
```

Worker receives messages:

```
self.addEventListener("message", function(event) {
 console.log("Worker received:", event.data)
})
```

Worker sends messages back:

```
self.addEventListener("message", function(event) {
 let result = event.data * 2
 self.postMessage(result)
})
```

Main script receives responses:

```
worker.addEventListener("message", function(event) {
 console.log("Main received:", event.data)
})
```

COMPLETE EXAMPLE

Main script:

```
let worker = new Worker("worker.js")

worker.addEventListener("message", function(event) {
 console.log("Result:", event.data)
})

worker.postMessage({ numbers: [1, 2, 3, 4, 5] })
```

Worker script:

```
self.addEventListener("message", function(event) {
 let numbers = event.data.numbers

 let sum = numbers.reduce(function(a, b) {
 return a + b
 }, 0)

 self.postMessage(sum)
})
```

HEAVY COMPUTATION EXAMPLE

Without worker (blocks UI):

```
function findPrimes(max) {
 let primes = []

 for (let num = 2; num <= max; num++) {
 let isPrime = true

 for (let i = 2; i <= Math.sqrt(num); i++) {
 if (num % i === 0) {
 isPrime = false
 break
 }
 }

 if (isPrime) {
 primes.push(num)
 }
 }

 return primes
}
```

```
let result = findPrimes(1000000)
```

With worker:

Main script:

```
let worker = new Worker("prime-worker.js")

worker.postMessage(1000000)

worker.addEventListener("message", function(event) {
 console.log("Found", event.data.length, "primes")
})

console.log("This runs immediately!")
```

prime-worker.js:

```
self.addEventListener("message", function(event) {
 let max = event.data
 let primes = []

 for (let num = 2; num <= max; num++) {
 let isPrime = true

 for (let i = 2; i <= Math.sqrt(num); i++) {
 if (num % i === 0) {
 isPrime = false
 break
 }
 }

 if (isPrime) {
 primes.push(num)
 }
 }

 self.postMessage(primes)
})
```

The main thread stays responsive while worker computes.

TERMINATING WORKERS

From main script:

worker.terminate()

From worker:

self.close()

ERROR HANDLING

Main script:

```
worker.addEventListener("error", function(event) {
 console.log("Worker error:", event.message)
 console.log("File:", event.filename)
 console.log("Line:", event.lineno)
})
```

Worker can throw errors:

```
self.addEventListener("message", function(event) {
 if (event.data < 0) {
 throw new Error("Negative numbers not allowed")
 }
})
```

WORKER LIMITATIONS

Workers cannot access:

- DOM (document, window)
- Parent page variables
- localStorage (in some browsers)

Workers can access:

- fetch API
- IndexedDB
- setTimeout, setInterval
- Math, JSON, Date
- Their own postMessage

INLINE WORKERS

Create worker from a function:

```
function createWorker(fn) {
 let blob = new Blob(
 ["(" + fn.toString() + ")()"],
 { type: "application/javascript" }
 )

 let url = URL.createObjectURL(blob)
 return new Worker(url)
}

let worker = createWorker(function() {
 self.addEventListener("message", function(event) {
 let result = event.data * 2
 self.postMessage(result)
 })
})
```

```
worker.postMessage(21)
```

TRANSFERABLE OBJECTS

Transfer ownership of data (faster for large data):

```
let buffer = new ArrayBuffer(1024 * 1024)

worker.postMessage(buffer, [buffer])

console.log(buffer.byteLength)
```

After transfer, the buffer is empty in main script.

SHARED WORKERS

Shared by multiple pages:

```
let shared = new SharedWorker("shared-worker.js")

shared.port.start()

shared.port.postMessage("Hello")

shared.port.addEventListener("message", function(event) {
 console.log(event.data)
})
```

shared-worker.js:

```
let connections = []

self.addEventListener("connect", function(event) {
 let port = event.ports[0]
```

```
 connections.push(port)

 port.addEventListener("message", function(event) {
 connections.forEach(function(conn) {
 conn.postMessage("Broadcast: " + event.data)
 })
 })

 port.start()
})
```

PRACTICAL EXAMPLES

Image processing worker:

Main:

```
let worker = new Worker("image-worker.js")

let imageData = canvas.getContext("2d").getImageData(0, 0, width,
height)

worker.postMessage({ imageData, filter: "grayscale" })

worker.addEventListener("message", function(event) {
 canvas.getContext("2d").putImageData(event.data, 0, 0)
})
```

image-worker.js:

```
self.addEventListener("message", function(event) {
 let { imageData, filter } = event.data
 let data = imageData.data

 if (filter === "grayscale") {
 for (let i = 0; i < data.length; i += 4) {
 let avg = (data[i] + data[i + 1] + data[i + 2]) / 3
```

```
 data[i] = avg
 data[i + 1] = avg
 data[i + 2] = avg
 }
 }

 self.postMessage(imageData)
})
```

Sorting worker:

Main:

```
let worker = new Worker("sort-worker.js")

let largeArray = generateLargeArray(1000000)

worker.postMessage(largeArray)

worker.addEventListener("message", function(event) {
 console.log("Sorted array:", event.data)
})
```

sort-worker.js:

```
self.addEventListener("message", function(event) {
 let array = event.data
 array.sort(function(a, b) {
 return a - b
 })
 self.postMessage(array)
})
```

WORKER POOL

Reuse workers for multiple tasks:

```
class WorkerPool {
 constructor(workerScript, size = 4) {
 this.workers = []
 this.queue = []

 for (let i = 0; i < size; i++) {
 let worker = new Worker(workerScript)
 worker.busy = false
 this.workers.push(worker)
 }
 }

 run(data) {
 return new Promise((resolve, reject) => {
 let task = { data, resolve, reject }

 let worker = this.workers.find(w => !w.busy)

 if (worker) {
 this.runTask(worker, task)
 } else {
 this.queue.push(task)
 }
 })
 }

 runTask(worker, task) {
 worker.busy = true

 let handler = (event) => {
 worker.removeEventListener("message", handler)
 worker.busy = false
 task.resolve(event.data)

 if (this.queue.length > 0) {
 this.runTask(worker, this.queue.shift())
 }
 }

 worker.addEventListener("message", handler)
```

```
 worker.postMessage(task.data)
 }
}

let pool = new WorkerPool("compute-worker.js", 4)

let results = await Promise.all([
 pool.run(data1),
 pool.run(data2),
 pool.run(data3),
 pool.run(data4)
])
```

PRACTICE

Create a worker that performs a calculation.

Send and receive messages between main and worker.

Move heavy computation to a worker.

Handle errors from workers.

Create a worker pool for parallel tasks.

WHAT YOU LEARNED

You learned workers run JavaScript in background threads.
You learned postMessage sends data to workers.
You learned addEventListener receives messages.
You learned workers cannot access the DOM.
You learned transferable objects for large data.
You learned worker pools for parallel processing.

The next chapter brings everything together with practice programs.

BOOK 9: ASYNC JAVASCRIPT

Chapter 10: Practice Async Programs

WEATHER APP

Fetch and display weather data:

HTML:

```
<div id="weather-app">
 <input type="text" id="city" placeholder="Enter city name">
 <button id="search">Get Weather</button>
 <div id="weather-display"></div>
 <div id="history"></div>
</div>
```

JavaScript:

```
let cityInput = document.getElementById("city")
let searchBtn = document.getElementById("search")
let weatherDisplay = document.getElementById("weather-display")
let historyDisplay = document.getElementById("history")

let apiKey = "YOUR_API_KEY"
let searchHistory =
JSON.parse(localStorage.getItem("weatherHistory")) || []

async function getWeather(city) {
 weatherDisplay.innerHTML = "<p>Loading...</p>"

 try {
 let response = await fetch(
`https://api.openweathermap.org/data/2.5/weather?q=${city}
&appid=${apiKey}&units=metric`
```

```
 )

 if (!response.ok) {
 throw new Error("City not found")
 }

 let data = await response.json()
 displayWeather(data)
 addToHistory(city)
 } catch (error) {
 weatherDisplay.innerHTML = `<p class="error">
${error.message}</p>`
 }
}

function displayWeather(data) {
 weatherDisplay.innerHTML = `
 <h2>${data.name}, ${data.sys.country}</h2>
 <p class="temp">${Math.round(data.main.temp)}C</p>
 <p class="desc">${data.weather[0].description}</p>
 <p>Humidity: ${data.main.humidity}%</p>
 <p>Wind: ${data.wind.speed} m/s</p>
 `
}

function addToHistory(city) {
 if (!searchHistory.includes(city)) {
 searchHistory.unshift(city)
 searchHistory = searchHistory.slice(0, 5)
 localStorage.setItem("weatherHistory",
JSON.stringify(searchHistory))
 renderHistory()
 }
}

function renderHistory() {
 historyDisplay.innerHTML = searchHistory.map(function(city) {
 return `<button class="history-btn">${city}</button>`
 }).join("")
}
```

```
searchBtn.addEventListener("click", function() {
 let city = cityInput.value.trim()
 if (city) {
 getWeather(city)
 }
})

cityInput.addEventListener("keypress", function(event) {
 if (event.key === "Enter") {
 let city = cityInput.value.trim()
 if (city) {
 getWeather(city)
 }
 }
})

historyDisplay.addEventListener("click", function(event) {
 if (event.target.classList.contains("history-btn")) {
 getWeather(event.target.textContent)
 }
})

renderHistory()
```

NEWS FEED WITH INFINITE SCROLL

Load articles as user scrolls:

HTML:

```
<div id="news-feed">
 <div id="articles"></div>
 <div id="loader">Loading more articles...</div>
</div>
```

JavaScript:

```
let articlesContainer = document.getElementById("articles")
let loader = document.getElementById("loader")

let page = 1
let isLoading = false
let hasMore = true

async function loadArticles() {
 if (isLoading || !hasMore) return

 isLoading = true
 loader.style.display = "block"

 try {
 await delay(500)

 let articles = await fetchArticles(page)

 if (articles.length === 0) {
 hasMore = false
 loader.textContent = "No more articles"
 } else {
 renderArticles(articles)
 page++
 }
 } catch (error) {
 console.error("Failed to load articles:", error)
 loader.textContent = "Failed to load. Try again."
 } finally {
 isLoading = false
 if (hasMore) {
 loader.style.display = "none"
 }
 }
}

async function fetchArticles(page) {
 let response = await fetch(`/api/articles?page=${page}
&limit=10`)
```

```
 if (!response.ok) {
 throw new Error("Failed to fetch articles")
 }

 return response.json()
}

function renderArticles(articles) {
 articles.forEach(function(article) {
 let articleEl = document.createElement("article")
 articleEl.innerHTML = `
 <h2>${article.title}</h2>
 <p class="meta">${article.author} - ${formatDate(article.date)}
</p>
 <p>${article.summary}</p>
 <a href="${article.url}">Read more</a>
 `
 articlesContainer.appendChild(articleEl)
 })
}

function formatDate(dateString) {
 return new Date(dateString).toLocaleDateString()
}

function delay(ms) {
 return new Promise(function(resolve) {
 setTimeout(resolve, ms)
 })
}

function checkScroll() {
 let scrollPosition = window.scrollY + window.innerHeight
 let documentHeight = document.documentElement.scrollHeight

 if (scrollPosition >= documentHeight - 300) {
 loadArticles()
 }
}
```

```
window.addEventListener("scroll", checkScroll)

loadArticles()
```

CHAT APPLICATION

Real-time messaging with polling:

HTML:

```
<div id="chat-app">
 <div id="messages"></div>
 <form id="message-form">
 <input type="text" id="message-input" placeholder="Type a
message...">
 <button type="submit">Send</button>
 </form>
</div>
```

JavaScript:

```
let messagesContainer = document.getElementById("messages")
let messageForm = document.getElementById("message-form")
let messageInput = document.getElementById("message-input")

let lastMessageId = 0
let username = prompt("Enter your username:")

async function sendMessage(text) {
 try {
 let response = await fetch("/api/messages", {
 method: "POST",
 headers: { "Content-Type": "application/json" },
 body: JSON.stringify({
 username,
 text,
```

```
 timestamp: Date.now()
 })
 })

 if (!response.ok) {
 throw new Error("Failed to send message")
 }
 } catch (error) {
 console.error("Send failed:", error)
 showError("Message failed to send")
 }
}

async function fetchNewMessages() {
 try {
 let response = await fetch(`/api/messages?after=
${lastMessageId}`)
 let messages = await response.json()

 if (messages.length > 0) {
 renderMessages(messages)
 lastMessageId = messages[messages.length - 1].id
 scrollToBottom()
 }
 } catch (error) {
 console.error("Fetch failed:", error)
 }
}

function renderMessages(messages) {
 messages.forEach(function(message) {
 let messageEl = document.createElement("div")
 messageEl.className = message.username === username ?
"message own" : "message"
 messageEl.innerHTML = `
 <span class="username">${message.username}</span>
 <span class="text">${message.text}</span>
 <span class="time">${formatTime(message.timestamp)}</
span>
 `
```

```
 messagesContainer.appendChild(messageEl)
 })
}

function formatTime(timestamp) {
 return new Date(timestamp).toLocaleTimeString()
}

function scrollToBottom() {
 messagesContainer.scrollTop = messagesContainer.scrollHeight
}

function showError(message) {
 let errorEl = document.createElement("div")
 errorEl.className = "error"
 errorEl.textContent = message
 messagesContainer.appendChild(errorEl)

 setTimeout(function() {
 errorEl.remove()
 }, 3000)
}

messageForm.addEventListener("submit", async function(event) {
 event.preventDefault()

 let text = messageInput.value.trim()
 if (!text) return

 messageInput.value = ""
 await sendMessage(text)
 await fetchNewMessages()
})

setInterval(fetchNewMessages, 2000)

fetchNewMessages()
```

FILE UPLOADER WITH QUEUE

Upload multiple files with progress:

HTML:

```
<div id="uploader">
 <input type="file" id="file-input" multiple>
 <div id="upload-queue"></div>
</div>
```

JavaScript:

```
let fileInput = document.getElementById("file-input")
let queueContainer = document.getElementById("upload-queue")

let uploadQueue = []
let isUploading = false

fileInput.addEventListener("change", function() {
 Array.from(this.files).forEach(function(file) {
 addToQueue(file)
 })

 this.value = ""
 processQueue()
})

function addToQueue(file) {
 let id = Date.now() + Math.random()

 let item = {
 id,
 file,
 status: "pending",
 progress: 0
 }

 uploadQueue.push(item)
```

```
 renderQueue()
}

async function processQueue() {
 if (isUploading) return

 let pending = uploadQueue.find(function(item) {
 return item.status === "pending"
 })

 if (!pending) return

 isUploading = true
 pending.status = "uploading"
 renderQueue()

 try {
 await uploadFile(pending)
 pending.status = "complete"
 } catch (error) {
 pending.status = "error"
 pending.error = error.message
 }

 isUploading = false
 renderQueue()
 processQueue()
}

function uploadFile(item) {
 return new Promise(function(resolve, reject) {
 let xhr = new XMLHttpRequest()

 xhr.upload.addEventListener("progress", function(event) {
 if (event.lengthComputable) {
 item.progress = Math.round((event.loaded / event.total) * 100)
 renderQueue()
 }
 })
```

```
 xhr.addEventListener("load", function() {
 if (xhr.status >= 200 && xhr.status < 300) {
 resolve()
 } else {
 reject(new Error("Upload failed"))
 }
 })

 xhr.addEventListener("error", function() {
 reject(new Error("Network error"))
 })

 let formData = new FormData()
 formData.append("file", item.file)

 xhr.open("POST", "/api/upload")
 xhr.send(formData)
 })
}

function renderQueue() {
 queueContainer.innerHTML = uploadQueue.map(function(item) {
 let statusClass = item.status
 let progressBar = ""

 if (item.status === "uploading") {
 progressBar = `<div class="progress-bar" style="width:
${item.progress}%"></div>`
 }

 return `
 <div class="queue-item ${statusClass}">
 <span class="filename">${item.file.name}</span>
 <span class="status">${item.status}</span>
 ${progressBar}
 ${item.error ? `<span class="error">${item.error}</span>` :
""}
 </div>
 `
 }).join("")
```

```
}
```

COUNTDOWN TIMER

Async countdown with notifications:

HTML:

```
<div id="timer-app">
 <input type="number" id="minutes" value="5" min="1">
 <button id="start">Start Timer</button>
 <div id="display">00:00</div>
 <button id="pause" disabled>Pause</button>
 <button id="reset" disabled>Reset</button>
</div>
```

JavaScript:

```
let minutesInput = document.getElementById("minutes")
let startBtn = document.getElementById("start")
let pauseBtn = document.getElementById("pause")
let resetBtn = document.getElementById("reset")
let display = document.getElementById("display")

let totalSeconds = 0
let remainingSeconds = 0
let intervalId = null
let isPaused = false

function formatTime(seconds) {
 let mins = Math.floor(seconds / 60)
 let secs = seconds % 60
 return `${mins.toString().padStart(2, "0")}:
${secs.toString().padStart(2, "0")}`
}

function updateDisplay() {
```

```
 display.textContent = formatTime(remainingSeconds)

 if (remainingSeconds <= 10 && remainingSeconds > 0) {
 display.classList.add("warning")
 } else {
 display.classList.remove("warning")
 }
}

async function startTimer() {
 totalSeconds = parseInt(minutesInput.value) * 60
 remainingSeconds = totalSeconds

 startBtn.disabled = true
 pauseBtn.disabled = false
 resetBtn.disabled = false
 minutesInput.disabled = true

 updateDisplay()

 intervalId = setInterval(function() {
 if (!isPaused) {
 remainingSeconds--
 updateDisplay()

 if (remainingSeconds <= 0) {
 clearInterval(intervalId)
 timerComplete()
 }
 }
 }, 1000)
}

async function timerComplete() {
 display.textContent = "Time's up!"
 display.classList.add("complete")

 if ("Notification" in window && Notification.permission ===
"granted") {
 new Notification("Timer Complete", {
```

```
 body: "Your timer has finished!"
 })
 }

 playSound()
 resetControls()
}

function playSound() {
 let context = new AudioContext()
 let oscillator = context.createOscillator()
 let gain = context.createGain()

 oscillator.connect(gain)
 gain.connect(context.destination)

 oscillator.frequency.value = 800
 gain.gain.setValueAtTime(0.5, context.currentTime)
 gain.gain.exponentialRampToValueAtTime(0.01,
context.currentTime + 0.5)

 oscillator.start(context.currentTime)
 oscillator.stop(context.currentTime + 0.5)
}

function pauseTimer() {
 isPaused = !isPaused
 pauseBtn.textContent = isPaused ? "Resume" : "Pause"
}

function resetTimer() {
 clearInterval(intervalId)
 remainingSeconds = 0
 display.textContent = "00:00"
 display.classList.remove("warning", "complete")
 resetControls()
}

function resetControls() {
 startBtn.disabled = false
```

```
 pauseBtn.disabled = true
 resetBtn.disabled = true
 minutesInput.disabled = false
 isPaused = false
 pauseBtn.textContent = "Pause"
}

startBtn.addEventListener("click", startTimer)
pauseBtn.addEventListener("click", pauseTimer)
resetBtn.addEventListener("click", resetTimer)

if ("Notification" in window && Notification.permission !==
"granted") {
 Notification.requestPermission()
}
```

WHAT YOU BUILT

You built five complete programs:

Weather app with API calls and local storage
News feed with infinite scroll
Chat application with polling
File uploader with queue and progress
Countdown timer with notifications

Each program demonstrated:

fetch API for HTTP requests
Promise handling with async/await
Error handling and user feedback
Loading states and progress indicators
Real-world async patterns

WHAT COMES NEXT

You now understand asynchronous JavaScript:

- Async concepts and timers (Book 9, Chapter 1)
- Callbacks (Book 9, Chapter 2)
- Promises (Book 9, Chapter 3)
- Async/await (Book 9, Chapter 4)
- Fetch API (Book 9, Chapter 5)
- Error handling (Book 9, Chapter 6)
- Async patterns (Book 9, Chapter 7)
- Real-world applications (Book 9, Chapter 8)
- Web Workers (Book 9, Chapter 9)

Book 10 teaches you error handling in depth. You will:

- Handle all types of errors
- Create custom error types
- Debug effectively
- Write robust code

Errors happen. Handling them well makes the difference.

You finished Book 9. You can work with async code. Now we master error handling.